

# Computational Electrodynamics with Chombo

Zdzislaw Meglicki

Indiana University


A Work in Progress Report

Prepared for the Argonne National Laboratory

April 23, 2004

*Document version* \$Id: AMR.tex,v 1.23 2004/04/23 21:10:45 meglicki Exp \$

## Abstract

This report describes in some detail a Chombo implementation of an adaptive mesh refinement (AMR) finite-difference time-domain (FDTD) demonstration code. The code simulates propagation of a Gaussian pulse in vacuum. The modeled system has been maximally simplified, so as not to obscure the AMR manipulations, which are the focus of the example. And so we have no PML at the boundary of the region and, consequently, time stepping is limited to 50 steps at the coarse level. The AMR structure is limited to two levels. Gnuplot animations generated by the program illustrate propagation of electric and magnetic fields at both AMR levels, as well as AMR cell tagging and evolution of the fine level grid structure. Open issues are flagged with the  sign on the margins. They are discussed, wherever there is anything constructive to say about them and related further development work is suggested.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Chombo</b>	<b>3</b>
<b>3</b>	<b>The Demo</b>	<b>7</b>
<b>4</b>	<b>The FDTD/AMR Time Step</b>	<b>14</b>
<b>5</b>	<b>The Code</b>	<b>17</b>
5.1	Constructing the Level 0 Grid and Data . . . . .	18
5.2	Preiteration Procedures . . . . .	20
5.3	The Iteration Loop . . . . .	21
5.4	Tagging Cells of Level 0 . . . . .	25
5.5	Constructing the Level 1 Grid . . . . .	27
5.6	Filling the Level 1 Grid With Data . . . . .	30
5.7	Advancing the Level 1 <b><i>E</i></b> . . . . .	32
5.8	Correcting the Level 0 <b><i>E</i></b> . . . . .	33
5.9	Generating Gnuplot Data Files . . . . .	34
5.10	Reading the Input File . . . . .	34
<b>6</b>	<b>Refluxing</b>	<b>35</b>
<b>7</b>	<b>A Multilevel System</b>	<b>36</b>
	<b>References</b>	<b>37</b>
	<b>Index</b>	<b>38</b>

# 1 Introduction

This document discusses in some detail implementation of a very simple adaptive mesh refinement (AMR) finite-difference time-domain [5] [6] (FDTD) demonstration code. The code is implemented using the Chombo package [1] developed at the Lawrence Berkeley National Laboratory (LBL) by the Applied Numerical Algorithms Group (ANAG).

The purpose of the code is to explore and demonstrate use of Chombo and AMR more than to solve a specific problem of computational electrodynamics. Consequently the electrodynamic part of the code, encapsulated on a separate Fortran file, has been simplified to an absolute minimum. The Fortran file is only about 100 lines long compared to about 1000 lines of the Chombo code. It implements simple propagation of a Gaussian pulse in 2 dimensions without absorption at the boundary. The cylindrical wave spreads symmetrically from the origin of the pulse in the middle of the computational domain. But the code does not make use of the symmetry of the problem. In effect the symmetry and its maintenance throughout the evolution of the system serve as an additional check on the correctness of the numerical procedures employed.

The program does not utilize Chombo IO subsystem, instead writing computed data as plain text files, formatted for display with Gnuplot [3]. Gnuplot command files for the display of the data are generated by the program too. This way the program's results can be viewed on a simple UNIX/Linux or Cygwin workstation without having to install additional visualization software (or hardware). The program itself can be compiled and run on the same workstation, if Chombo has been installed on it too.

At this stage of the development and research I prefer to stay away from parallel computer systems and from 3-dimensional computations, focusing instead on the FDTD/AMR algorithm, its basic implementation within the Chombo framework, and the correctness of numerical procedures used by the program. It is very important that these basic problems are fully resolved first, before we embark on much more complicated large scale production work.

The moment we switch to 3-dimensional computations we will have to embrace Chombo parallelization, Chombo IO and Chombo visualization all at the same time. The complexity of the simulations at that stage and the required logistics will become so overwhelming that the study of algorithmic correctness and fundamental issues will become markedly more difficult.

## 2 Chombo

Chombo is a library of C++ classes and functions for implementing and managing rather complex logistics and data movements associated with adaptively refined grids used in parallel finite difference codes. The basic idea behind Chombo is that all such manipulations can be carried out within a Chombo C++ wrapper, with the actual finite difference computations offloaded onto a simple Fortran subroutine.

This means that a Chombo user must be equally comfortable with both C++ and Fortran, as well as with subtleties of calling Fortran subroutines from a C++ program. The way Chombo data is laid out in the computer's memory *assumes* that it is going to be passed to Fortran and not to a C or C++ function.

Although it would be possible to do it all either in C++ or in Fortran, for neither of the two languages is significantly superior to the other, this dichotomy reflects differing preferences of computer and computational scientists. Where the two end up working with each other, a hybrid C++/Fortran child is born.

Chombo can be downloaded from

<http://seesar.lbl.gov/anag/chombo/>

Chombo is not a stand-alone code. It depends on other software, most notably on GNU make, Perl, HDF5 [4] and MPI. C++ and Fortran-77 compilers are required too. Specific versions of third party software packages are listed on the Chombo WWW page. Some of these are optional. For example, it is possible to configure and compile a non-MPI, non-HDF5 version of Chombo.

Chombo installation itself may be a little daunting, and generally users should probably request that their systems administrators do it for them. Chombo should be configured and compiled separately for its various flavours, e.g., 2-dimensional versus 3-dimensional, parallel versus sequential, double precision versus single precision, with HDF5 versus without and so on. On the other hand, the package is not as large or computationally heavy as it seems to be at first glance, and it is perfectly possible to run 2-dimensional double precision Chombo programs on a laptop. The demonstration program discussed in this report was developed and tested on an 863 MHz Pentium III with 256 MB memory and a 30 GB internal drive under Windows XP/Cygwin. A rather unimpressive system by today's standards.

Cygwin or MacOS X installation of Chombo may be somewhat tricky on account of Windows and Macintosh file systems not distinguishing between upper and lower case letters in file names. There is a manual cludge around this problem – but here, again, it would be best for the user to seek professional assistance.

Chombo installation is discussed in the “Chombo Design Document” [2], which can be downloaded from the Chombo WWW site. Additionally the Chombo source is provided with the Chombo Reference manual in the HTML format, the first page of which lives on

`Chombo/lib/doc/doxygen/html/index.html`

The Design Document fulfills the role of a “User Guide” and explains the general concepts and programmer interfaces to various Chombo utilities. But it is not an easy document to read and it is not complete either. Chombo source is provided with several examples, mostly rather elaborate, and their study is essential for anyone trying to master the package.

Chombo is compiled and installed *in situ*, which means that its libraries and includes are built during the compilation and placed right above the source in the top level Chombo library directory. There is no “make install” step common in other software packages.

The top level Chombo directory splits into two branches:

`Chombo/example`  
`Chombo/lib`

The `example` directory tree contains some of the examples mentioned above. Other examples can be found in the `lib/test` subdirectory.

The `lib` directory tree of Chombo contains the following subdirectories:

`Chombo/lib/doc`  
`Chombo/lib/include`  
`Chombo/lib/mk`  
`Chombo/lib/src`  
`Chombo/lib/test`

The reference manual lives in the `doc` subdirectory, as I have mentioned above. The `include` subdirectory contains Chombo includes. There are nearly a 100 of them. Every major Chombo class comes with its own include file. The `mk` subdirectory contains GNU make includes. These files encapsulate a lot of Chombo complexity, so that `GNUmakefile` that a user may have in her or his Chombo program directory can be deceptively simple. The `src` subdirectory contains the Chombo source and the `test` subdirectory contains several tests that are run during the system installation to make sure that everything is going to work. The tests are worth studying in their own right. They are markedly less complex than the examples provided in the `Chombo/example` directory tree.

Once Chombo has been compiled and installed the library files are placed in the `Chombo/lib` directory. There is a separate library for every major Chombo toolkit. The libraries have very long names that encapsulate not only what the library is for, but also the architecture the library has been compiled for as well as other configuration details. For example, on my system, which is Windows XP/Cygwin, and on which I used GNU g++ and g77 compilers to construct a 2-dimensional, double precision version of Chombo, the libraries have the following names:

```
libamrelliptic2d.CYGWIN.g++.g77.a
libamrtimedependent2d.CYGWIN.g++.g77.a
libamrtools2d.CYGWIN.g++.g77.a
libarrayview2d.CYGWIN.g++.g77.a
libboxtools2d.CYGWIN.g++.g77.a
libparticletools2d.CYGWIN.g++.g77.a
```

But a user is *not* expected to link against these libraries explicitly. The Chombo make file system will look up appropriate includes and load appropriate libraries automatically.

Here is an example of how it all comes together. The code this report is going to talk about lives in a directory called `AMR`. There are three files in this directory:

```
AMR/GNUmakefile
AMR/fdtd.cpp
AMR/update.F
```

The `fdtd.cpp` file contains the C++ Chombo code. The `update.F` file contains the Fortran-77 code that is used to timestep the system and `GNUmakefile` contains make instructions.

This is what `GNUmakefile` looks like:

```
CHOMBO_HOME = /home/ANL/src/Chombo/lib
ebase := fdtd
LibNames := BoxTools AMRTools
BASE_DIR = .
VPATH_LOCAL = .
F_SOURCES =
CXX_SOURCES = fdtd.cpp
FORT_SOURCES = update.F
```

```

C_SOURCES =
INPUT = fdtd.input
include $(CHOMBO_HOME)/mk/Make.example

```

The first line defines the location of the Chombo library, `CHOMBO_HOME`. The second line, `ebase`, defines the *base* for the name of the binary to be built. The binary in this case is going to be called

```
fdtd2d.CYGWIN.g++.g77.ex
```

The program will need Box Tools and AMR Tools. This is specified on the third line. The `CXX_SOURCES` and `FORT_SOURCES` lines specify where C++ and Fortran-77 sources are. Finally `GNUmakefile` includes `$(CHOMBO_HOME)/mk/Make.example`, that is going to do all the rest, i.e., point the compiler to the appropriate includes and load appropriate libraries.

To make the program the user types `make fdtd`, and this is how the Chombo make system is going to respond:

```

$ make fdtd
Depending update.F ...
Depending fdtd.cpp ...
make fdtd2d.CYGWIN.g++.g77.ex
make[1]: Entering directory '/home/ANL/src/demo'
g++ -O2 -funroll-loops -Wno-long-long -Wno-sign-compare \
    -Wno-deprecated -ftemplate-depth-25 -DCH_SPACEDIM=2 \
    -DCH_CYGWIN -DNDEBUG -DCH_USE_DOUBLE -I/home/ANL/include \
    -DHDF5 -DCH_FORT_UNDERSCORE -I/home/ANL/src/Chombo/lib/src/BoxTools \
    -I/home/ANL/src/Chombo/lib/src/AMRTools -DENABLE_MEMORY_TRACKING \
    -DCH_LANG_CC -c fdtd.cpp -o o/2d.CYGWIN.g++.g77/fdtd.o
g++ -E -P -DCH_SPACEDIM=2 -DCH_CYGWIN -DNDEBUG -DCH_USE_DOUBLE \
    -I/home/ANL/include -DHDF5 -DCH_FORT_UNDERSCORE \
    -I/home/ANL/src/Chombo/lib/src/BoxTools \
    -I/home/ANL/src/Chombo/lib/src/AMRTools -DENABLE_MEMORY_TRACKING \
    -DCH_LANG_FORT update.F \
    | perl /home/ANL/src/Chombo/lib/util/chfpp/fort72 \
    | awk 'NF>0' > f/2d.CYGWIN.g++.g77/update.f
g77 -O2 -funroll-loops -fno-second-underscore -malign-double \
    -c f/2d.CYGWIN.g++.g77/update.f -o o/2d.CYGWIN.g++.g77/update.o
g++ -O2 -funroll-loops -Wno-long-long -Wno-sign-compare -Wno-deprecated \
    -ftemplate-depth-25 o/2d.CYGWIN.g++.g77/fdtd.o \
    o/2d.CYGWIN.g++.g77/update.o -L/home/ANL/src/Chombo/lib \
    -lboxtools2d.CYGWIN.g++.g77 -lamrtools2d.CYGWIN.g++.g77 \
    -L/home/ANL/lib -lhdf5 -lz -lg2c -lfrtbegin -lm \
    -o fdtd2d.CYGWIN.g++.g77.ex \
    |& awk -f /home/ANL/src/Chombo/lib/mk/tempnam.awk
make[1]: Leaving directory '/home/ANL/src/demo'
$

```

The compilation and linking of a Chombo program process is not trivial, but the compilation process does not clutter the working directory. Instead various auxiliary files are left on subdirectories called **d**, **f**, **o**, and **p**. In particular, depend files live in the **d** directory. The **f** directory contains preprocessed Fortran code. The **o** directory contains object files and the **p** directory is going to be empty in this case. The binary, as I have mentioned already, is going to be written on **fdtd2d.CYGWIN.g++.g77.ex**.

To run the binary, the user simply types **make run**, and the Chombo make system figures out which binary to run, with what options, if any, and which input file to use:

```
$ make run
Running fdtd for configuration 2d.CYGWIN.g++.g77 ...
Tagging on differences.
Tagging on values.
Advancing Ez_1.
Advancing Hx_1 and Hy_1.
Correcting Ez_0.
Correcting Hx_0 and Hy_0.
Reusing level 1 E data.
Reusing level 1 H data.
Quiet run.
.....
.....
.....
peak memory usage: 1637544 bytes (1 Mb)
... fdtd finished with status 0

$
```

In this case the program runs silently, just printing a dot every time it completes an iteration. But numerous data files are written on the working directory too.

### 3 The Demo

The demonstration program discussed in this report solves a 2-dimensional version of Maxwell equations. Vacuum is assumed and there are no free charges or currents. The computational domain is a  $60 \times 60$  square. There are no absorbing boundary conditions either. In short the electromagnetic component of the code itself is kept as simple as possible.

The equations of motion that are solved by the code are therefore:

$$\frac{1}{c} \frac{\partial E_z}{\partial t} = \frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} \quad (1)$$

$$\frac{1}{c} \frac{\partial H_x}{\partial t} = -\frac{\partial E_z}{\partial y} \quad (2)$$

$$\frac{1}{c} \frac{\partial H_y}{\partial t} = \frac{\partial E_z}{\partial x} \quad (3)$$

The FDTD method distributes electric and magnetic fields on the sides of the grid and staggers them in time in such a way that the divergence equations are satisfied automatically.

Assuming a regular, rectangular grid with  $\Delta x = \Delta y$  and the time step of

$$\Delta t = \frac{\Delta x}{2c} \quad (4)$$

we end up with the following discretized field equations

$$\begin{aligned} E_z^{n+1/2}(i, j) - E_z^{n-1/2}(i, j) \\ = \frac{1}{2} \left( H_y^n \left( i + \frac{1}{2}, j \right) - H_y^n \left( i - \frac{1}{2}, j \right) - H_x^n \left( i, j + \frac{1}{2} \right) + H_x^n \left( i, j - \frac{1}{2} \right) \right) \end{aligned} \quad (5)$$

$$H_x^{n+1} \left( i, j + \frac{1}{2} \right) - H_x^n \left( i, j + \frac{1}{2} \right) = -\frac{1}{2} \left( E_z^{n+1/2}(i, j+1) - E_z^{n+1/2}(i, j) \right) \quad (6)$$

$$H_y^{n+1} \left( i + \frac{1}{2}, j \right) - H_y^n \left( i + \frac{1}{2}, j \right) = \frac{1}{2} \left( E_z^{n+1/2}(i+1, j) - E_z^{n+1/2}(i, j) \right) \quad (7)$$

Observe that neither  $x$ , nor  $y$ , nor  $t$ , nor  $dt$ , nor  $dx$  appear in these equations explicitly. This means that we can use the same simple Fortran code for time stepping both the coarse and the fine level, as long as we remember to reduce both  $dt$  and  $dx$  at the same rate when switching from the coarse to the fine level.

FDTD fields are distributed on a staggered grid, i.e.,  $E_z$ ,  $H_x$  and  $H_y$  are not defined on the same geometric points, or on the same time slices.  $E_z$  is defined on  $(i, j)$ , but  $H_x$  is defined on  $(i, j + 1/2)$ , whereas  $H_y$  is defined on  $(i + 1/2, j)$ . Similarly,  $E_z$  is defined on the  $n + 1/2$  time slices, whereas  $H_x$  and  $H_y$  are defined on the  $n$  slices.

Chombo provides some basic level support for computations on staggered grids. For example, when a Chombo *box*, i.e., a rectangular region of space filled with cubic cells, is defined, it is possible to specify, whether the grid points associated with the cells are centered on the cell volumes or on the cell sides, or on the cell corners. Various operations that are then performed on the box, such as enlargement, or refinement, or coarsening, will take the grid centering into account and act accordingly. But this support doesn't go far enough. When the so called *disjoint box layout* is defined, there is already an assumption there that the grids filling the boxes are cell centered. Chombo fields, referred to as *level data* are then defined on top of the *disjoint box layouts*.

The reason for this lack of support for face or node centered data at the higher levels of Chombo is that adjacent face centered boxes are *not* disjoint. They overlap, because they share the faces – and this causes problems.

But Chombo goes some way towards handling face and node centered data. There is a class called *NodeFArrayBox* that is designed to contain node centered data. It can be used to distribute such data over a disjoint box layout, but the box layout itself is still cell-centered and the *NodeFArrayBox* class is really only a wrapper around the cell-centered data container.

Although I have implemented a simple FDTD code that utilized face and node centered Chombo grids, the only Chombo tools that the code employed were Chombo boxes. This therefore remains an open problem, both for us and for Chombo designers, to provide full support for staggered grids within the Chombo framework.

But in our context it is easy enough to work around it, the more so as engineers who use FDTD in their work seldom make much fuss about the specific placement of the fields, preferring instead to hand-wire this information into a Fortran code wherever required. And so, equations (5), (6) and (7) simply get converted into the following lines of Fortran-77:

```
Ez(i, j) = Ez(i, j)
```



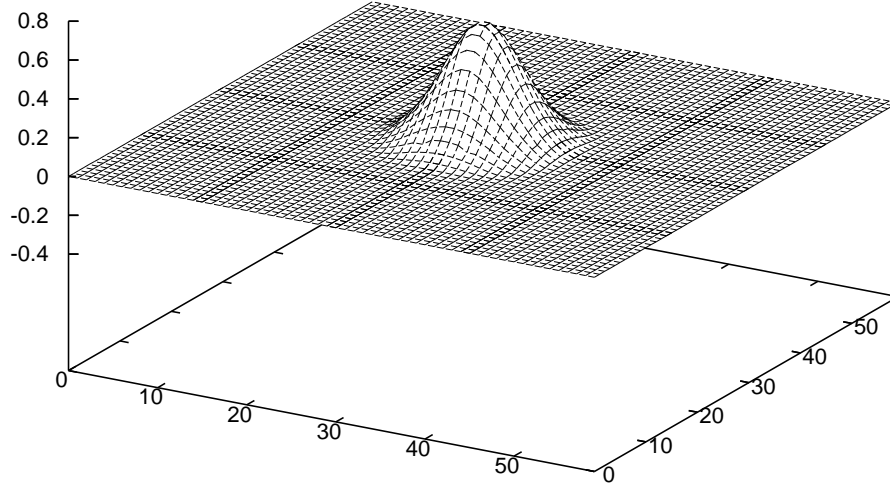


Figure 1: Field  $E_z$  at  $t = 3.0$ . Plot from level 0 data.

```
& + 0.5 * (Hy(i, j) - Hy(i - 1, j) - Hx(i, j) + Hx(i, j - 1))
```

and then

```
Hx(i, j) = Hx(i, j) + 0.5 * (Ez(i, j) - Ez(i, j + 1))
Hy(i, j) = Hy(i, j) + 0.5 * (Ez(i + 1, j) - Ez(i, j))
```

In other words,  $E_z$ ,  $H_x$  and  $H_y$  data are all represented by Fortran arrays of the same size – it is just that we have to remember that, e.g.,  $Hx(i, j)$  really means  $H_x(i, j + 1/2)$ ,  $Hx(i, j-1)$  really means  $H_x(i, j - 1/2)$ ,  $Hy(i, j)$  really means  $H_y(i + 1/2, j)$  and  $Hy(i-1, j)$  really means  $H_y(i - 1/2, j)$ .

This way we can forget, at least temporarily, about various placement of FDTD data and develop the Chombo application as if all this data was cell-centered. In this particular demonstration program we will not even have to remember about the trick, because absolute positioning is used in the  $\mathbf{E}$  advance part of the code only, and  $E_z$  is cell centered in 2D.

The initial condition for the electric and magnetic fields is zero. A Gaussian pulse (Gaussian in space and time) is then activated in the middle of the region. Figure 1 shows field  $E_z$  at  $t = 3.0$ .

The pulse, which overwrites  $E_z$  within the radius of 8.0 from the centre of the region at the end of every time step (such a source is called *hard*), is allowed to grow until  $t = 5.0$ , whereupon the field is allowed to relax freely, i.e., the overwrite is disabled at this stage. This results in the field bounce

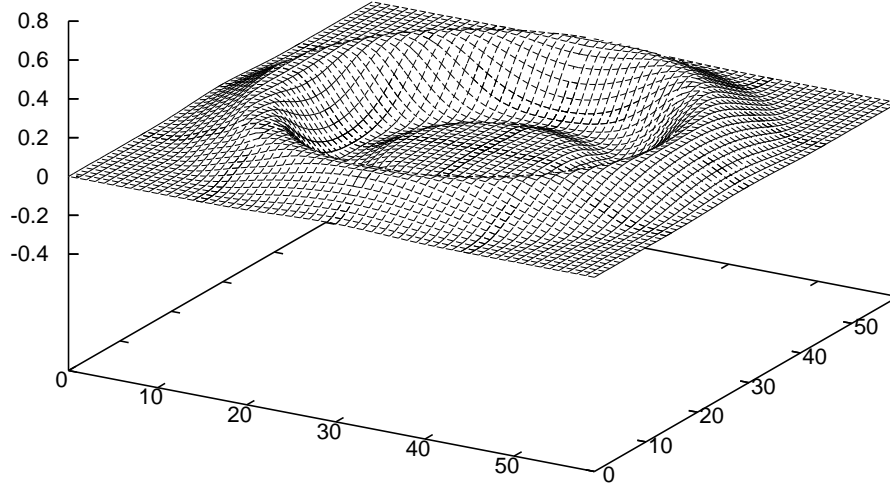


Figure 2: Field  $E_z$  at  $t = 23.0$ . Plot from level 0 data.

in the central region and a propagation of a cylindrical wave away from the centre. The resulting field configuration is shown in figure 2.

Because we have no absorbing boundary conditions, fields are propagated only until  $t = 25.0$ .

At every level 0 time step, we inspect field values in every level 0 cell. We now have two options for tagging the cells, depending on the absolute field values or on the values of field derivatives.

The first option checks if field values exceed a certain threshold in the cell. If they do, we tag the cell for refinement by adding it to the *set of tagged cells*.

The second option checks if field derivatives exceed a certain different threshold. If they do, we tag the cell for refinement in the same way.

The two options can be combined, i.e., cells can be tagged both on field values and field derivatives.

If the set of tagged cells is not empty at the end of this procedure, we generate dynamically level 1 that comprises disjoint boxes filled with cell centered grids, the grid constant of which is half of the level 0 grid constant, i.e.,

$$\Delta x_1 = \Delta x_0/2 \quad (8)$$

This level 1 structure is then populated with electric and magnetic fields, obtained by interpolating data from level 0 and the level 1 fields are then advanced in time with the time step halved compared to the level 0 time step, i.e.,

$$\Delta t_1 = \Delta t_0/2 \quad (9)$$

Figure 3 shows a set of tagged cells at  $t = 20.5$ . Cells are tagged on field values only in this example. Figure 4 shows a level 1 grid that is generated in this case.

Finally, figure 5 shows a portion of level 1 data for  $E_z$ .

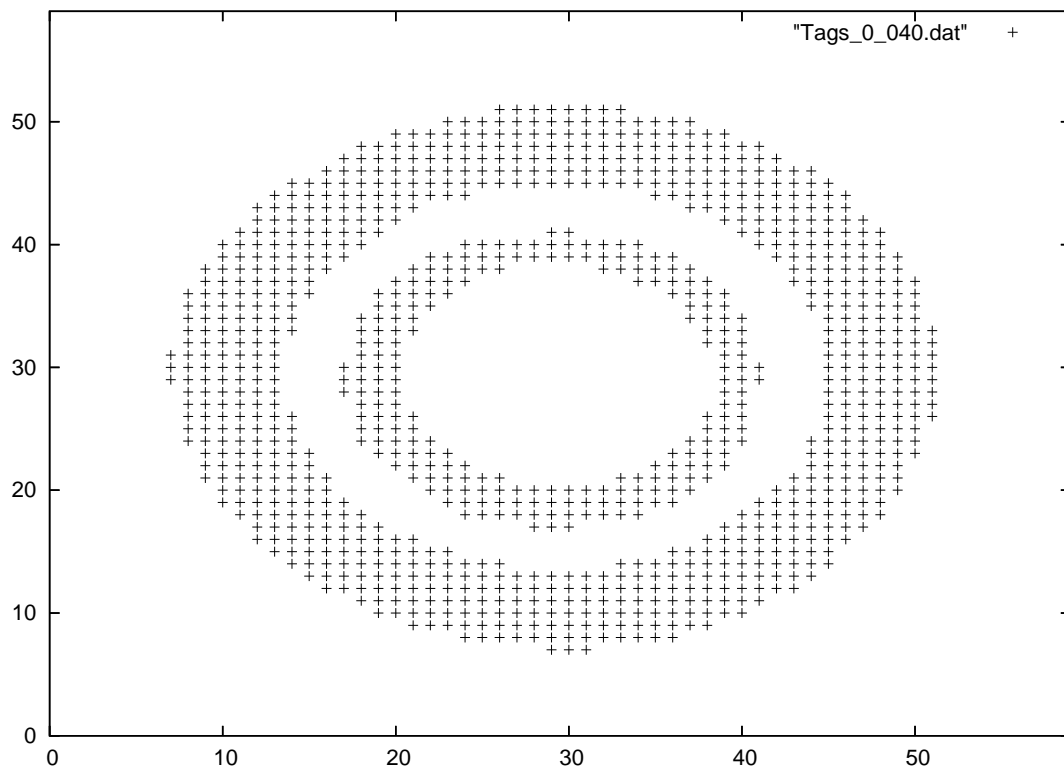


Figure 3: A set of tagged cells at  $t = 20.5$ .

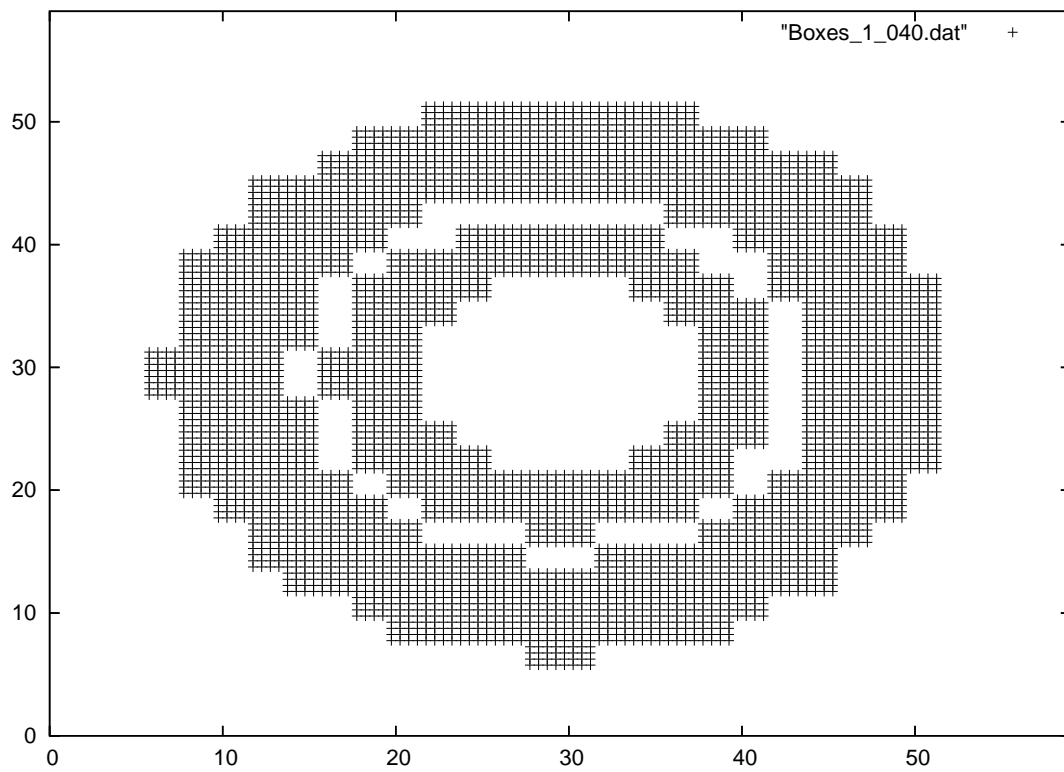


Figure 4: A level 1 grid at  $t = 20.5$ .

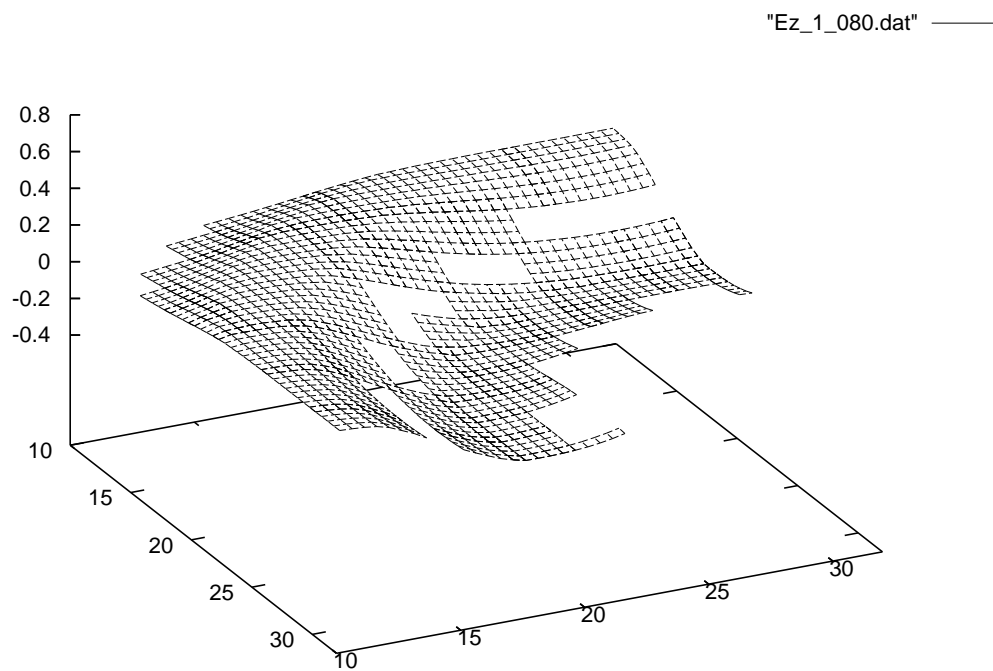


Figure 5:  $E_z$  at  $t = 20.5$ . Plot from level 1 data.

Animations that display evolution of  $E_z$ ,  $H_x$  and  $H_y$  at both levels, as well as evolution of the tagged cells set and of the level 1 grid can be displayed by typing the following commands:

```
$ gnuplot Ez_0.plt
$ gnuplot Hx_0.plt
$ gnuplot Hy_0.plt
$ gnuplot Ez_1.plt
$ gnuplot Hx_1.plt
$ gnuplot Hy_1.plt
$ gnuplot Tags_0.plt
$ gnuplot Boxes_1.plt
```

Selected data files can be also displayed on top of each other. For example, in order to see how level 1 boxes are built around the tagged cells the user should type the following commands to the gnuplot program:

```
gnuplot> set xrange[10:30]
gnuplot> set yrange[10:30]
gnuplot> plot "Tags_0_040.dat" with points, "Boxes_1_040.dat" with points
```

The resulting diagram is shown in figure 6.

Synchronized time stepping at both levels is quite complicated. To begin with level 0 time stepping must be always ahead of level 1 time stepping, and level 0 fields for previous time slices must be preserved so that boundary conditions can be constructed for level 1 fields at every level 1 time step. The level 1 boundary conditions are constructed by interpolating level 0 data both in space and in time.

At the end of each bi-level time step, level 0 data is corrected by replacing it with level 1 data averaged to the level 0 grid.

There is one more operation that must be carried out in order to complete the bi-level time step procedure. This operation is called *refluxing* and the current demonstration program doesn't do this yet. Chombo *does* provide special classes for flux correction and they are probably going to be sufficient. However, at this stage it is not clear to me, nor is it clear to Chombo designers, how the fluxes should be constructed in the FDTD leap-frog context, how they should be corrected, and how these corrections should then be translated into specific changes to the  $E_z$  and  $\mathbf{H}$  fields on the fine/coarse boundary. This is an open topic for further research.

I will discuss this issue in some more detail in section 6.

## 4 The FDTD/AMR Time Step

Figure 7 illustrates the basic leapfrog time stepping scheme of FDTD.

Given  $\mathbf{E}(t_0)$  and  $\mathbf{H}(t_0 + \Delta t/2)$  we compute  $\mathbf{E}(t_0 + \Delta t)$  and then we use this result to compute  $\mathbf{H}(t_0 + 3\Delta t/2)$ .

Figure 8 shows that straightforward time stepping the fine level in synchrony with the basic level for  $\Delta t_1 = \Delta t_0/2$  is impossible.



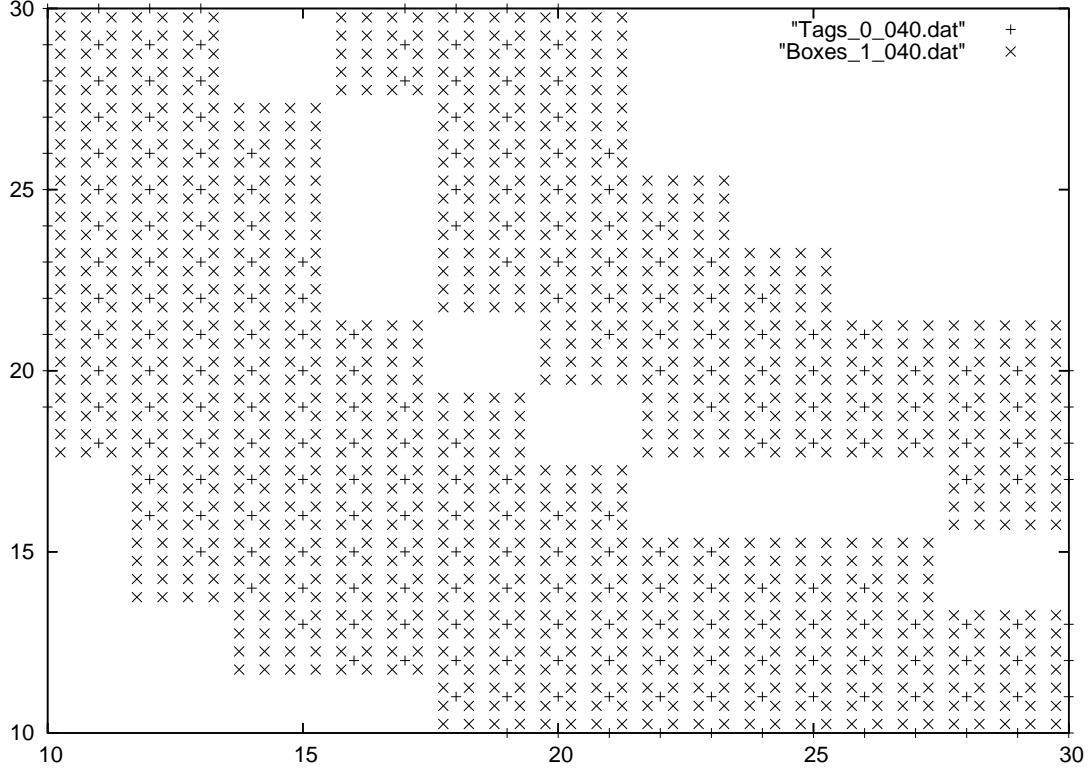


Figure 6: Tagged level 0 cells and level 1 grid points at  $t = 20.5$ .

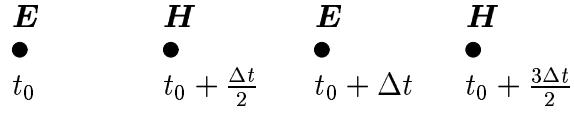


Figure 7: The basic leapfrog time stepping scheme of FDTD.

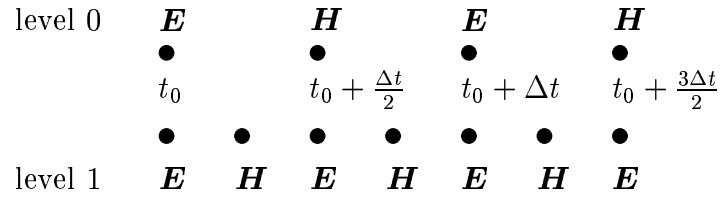


Figure 8: Two level FDTD leapfrog.  $\Delta t_1 = \Delta t_0/2$ .

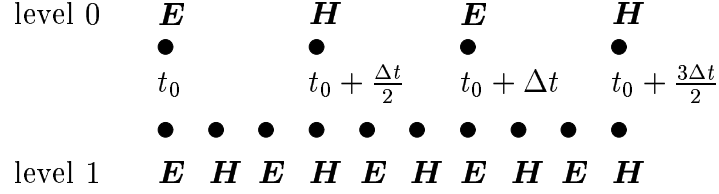


Figure 9: Two level FDTD leapfrog.  $\Delta t_1 = \Delta t_0/3$ .

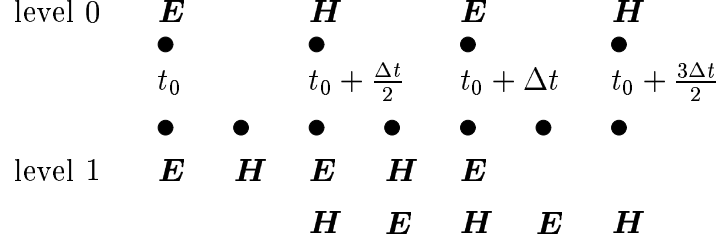


Figure 10: Two level FDTD leapfrog.  $\Delta t_1 = \Delta t_0/2$ . level 1 ***E*** and ***H*** are advanced separately.

We can sync both levels on the electric field, but then level 1 will be always  $\Delta t/2$  off on the magnetic field, and vice versa.

A possible solution here is to adopt  $\Delta t_1 = \Delta t_0/3$ , while preserving  $\Delta x_1 = \Delta x_0/2$ . This would not violate the CFL stability criterion, while at the same time we would be able to sync both ***E*** and ***H***, as shown in figure 9. The problem here is that computers are not very good at dividing by three.

Another solution is to stick to  $\Delta t_1 = \Delta t_0/2$ , but to time step level 1 ***E*** first between  $t_0$  and  $t_0 + \Delta t$ , then go back to  $t_0 + \Delta t/2$  and time step level 1 ***H*** between  $t_0 + \Delta t$  and  $t_0 + 3\Delta t/2$ . This is the solution that I have implemented in the demonstration code. Figure 10 illustrates this strategy.

It would be interesting to compare the two schemes, i.e., the one with  $\Delta t_1 = \Delta t_0/3$  and the one with  $\Delta t_1 = \Delta t_0/2$  and separate advances for level 1 ***E*** and ***H***. We would have to scrutinize whether the  $\Delta t_0/3$  scheme does not result in some unexpected problems, although at first glance there shouldn't be any, other than possible inaccuracies arising from dividing by 3. Chombo will not allow for  $\Delta x$  to be divided by 3, but in this case we would keep  $\Delta x_1 = \Delta x_0/2$ .

Time stepping level 1 data requires provision of boundary conditions to the level 1 region on the level 1/level 0 border. These conditions must be constructed by interpolating level 0 data *both* in space and in time. Figure 11 shows how level 0 data must be interpolated to construct boundary conditions for the level 1 ***E*** advance.

The lines connecting both levels illustrate data transfers. For example, in order to construct an initial condition for the level 1 ***E*** time step, we need to have ***H***( $t_0 + \Delta t/4$ ). We can construct it by interpolating between ***H***( $t_0 - \Delta t/2$ ) and ***H***( $t_0 + \Delta t/2$ ) at level 0. Similarly, in order to provide boundary conditions for ***E***( $t_0 + \Delta t/2$ ) at level 1, we have to interpolate data between ***E***( $t_0$ ) and ***E***( $t_0 + \Delta t$ ) at level 0, and so on.

Figure 12 shows interpolations required to advance ***H*** at level 1. Observe that this time we must use ***E***( $t_0 + 2\Delta t$ ) at level 0 in order to construct ***E***( $t_0 + 1.25\Delta t$ ) at level 1.

What all this adds up to is that level 1 ***E*** and ***H*** advance must be time-padded by level 0 data. In order to advance level 1 from  $t_0$  to  $t_0 + 1.5\Delta t$ , which is the time slice of ***H***, we must have level 0



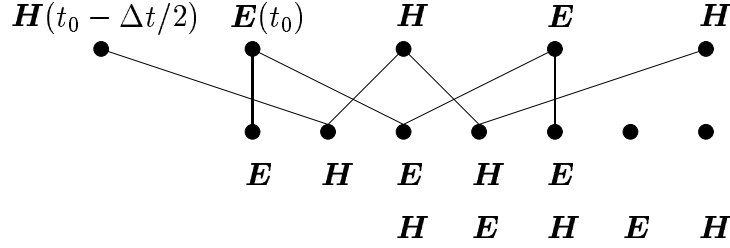


Figure 11: Interpolations from level 0 to level 1 for the level 1  $\mathbf{E}$  advancing procedure.  $\mathbf{H}(t_0 - \Delta t/2)$  from the previous time step must be remembered.

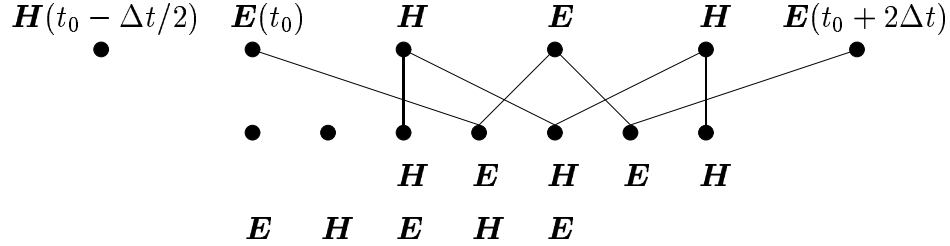


Figure 12: Interpolations from level 0 to level 1 for the level 1  $\mathbf{H}$  advancing procedure.  $\mathbf{E}(t_0 + 2\Delta t)$  at level 0 is needed to provide boundary conditions for  $\mathbf{E}(t_0 + 1.25\Delta t)$  at level 1.

data for all times between  $t_0 - \Delta t/2$  (the time slice of  $\mathbf{H}$ ) to  $t_0 + 2\Delta t$  (the time slice of the *next*  $\mathbf{E}$ ). This time padding is needed for both level 1 time stepping schemes considered in this section.

## 5 The Code

In this section I am going to explain the code itself, as well as various algorithmic details, which should be easier to understand when they are presented side by side with the code.

The C++ portion of the code lives just one one file. I tried to keep everything together so that the code would be more concise and easier to read. The code begins with definitions or various functions, which are all relatively trivial and its `main` portion is then presented towards the end of the file and is only about 600 lines long.

The basic structure of `main` is as follows:

```
Construct a level 0 grid
Define data (E and H) on the level 0 grid
Prepare auxiliary counters and registers used in the iteration
Iterate
Write Gnuplot command files.
```

Within the `Iterate` portion of the code we time step level 0, and then generate level 1 and time step it too. Data pertaining to both levels is written on Gnuplot data files. The structure of the `Iterate` portion of the code is as follows:

```

Begin Iterate
  Update Ez at level 0
  Update Hx and Hy at level 0
  Update Ez at level 0 again (to get Ez(t + 1.5 * dt))
  Tag level 0 cells
  if (set of tags is not empty) then
    Generate a level 1 grid
    Prepare devices for generating level 1 data from level 0 data
    Prepare initial data for advancing Ez at level 1
    Write Gnuplot data for Ez at level 1
    Update Ez at level 1
    Write Gnuplot data for Ez at level 1
    Update Hx and Hy at level 1
    Update Ez at level 1
    Correct Ez at level 0 using level 1 data
    Prepare initial data for advancing Hx and Hy at level 1
    Write Gnuplot data for Hx and Hy at level 1
    Update Hx and Hy at level 1
    Write Gnuplot data for Hx and Hy at level 1
    Update Ez at level 1
    Update Hx and Hy at level 1
    Correct Hx and Hy at level 0 using level 1 data
  end if
  Write Gnuplot data for Ez, Hx and Hy at level 0 for this time slice
  Save a pre-update Hx and Hy for use in the next iteration
End Iterate

```

There are numerous subtleties and details at various places, which this basic outline of the code structure doesn't cover, but I'm going to explain them all as we go along. This should make the general structure of the code easier to understand too.

## 5.1 Constructing the Level 0 Grid and Data

The level 0 grid comprises 16 boxes that cover the  $60 \times 60$  domain. It is not necessary, in principle, to split level 0, but by doing so I demonstrate how computations can be parallelized, even computations that pertain to level 0 alone. The way Chombo distributes the load is to assign each box and computations that unfold within it to a different processor. Chombo provides methods for exchange of ghost cells data between the boxes.

But before we can cover the domain with the boxes, we have to define it in the first place. This is done as follows:

```

#define NX 60
#define REFINERATIO 2
...
int number_of_cells_0 = NX;
int number_of_cells_1 = number_of_cells_0 * REFINERATIO;

```

```

ProblemDomain
    domain_0(IntVect::Zero, (number_of_cells_0 - 1) * IntVect::Unit);
ProblemDomain
    domain_1 = refine(domain_0, REFINE_RATIO);

```

The class `ProblemDomain` implements a uniform rectangular grid and adds to it information about boundary conditions, i.e., periodic versus non-periodic. Non-periodic boundary conditions are a default, and so we don't have to specify them here explicitly. The domain is specified by providing coordinates of the lower-left corner as the first argument, and coordinates of the upper-right corner as the second argument. The class `IntVect` implements simple vector objects with integer coordinates. The specific constants `IntVect::Zero` and `IntVect::Unit` correspond to  $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$  and  $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$  respectively. `IntVect` vectors can be multiplied by numbers, which is what we do here to generate the upper right corner coordinates.

A friend function `refine` is used here to generate the level 1 domain out of the level 0 domain that has just been defined.

The next portion of the code splits the level 0 domain into 16 boxes and organizes them into a *disjoint box layout*. Although you can define data directly on a box, which can be extracted from a domain (a domain is basically a box plus boundary conditions), data so defined will not be distributed across the CPU cluster. So we have to go through the procedure of first defining the disjoint box layout and then of defining level data on the layout to end up with *distributed* data.

Here is how the disjoint box layout for level 0 is constructed:

```

int number_of_boxes_0 = NB;
int box_size_0 = number_of_cells_0 / number_of_boxes_0;
Vector<Box> vector_of_boxes_0;
domainSplit(domain_0, vector_of_boxes_0, box_size_0);
Vector<int> vector_of_processes_0;
LoadBalance(vector_of_processes_0, vector_of_boxes_0);
DisjointBoxLayout box_layout_0(vector_of_boxes_0, vector_of_processes_0);
DataIterator data_iterator_0(box_layout_0);

```

The first step is to define a `vector_of_boxes`, which is what it says that it is. It is a vector of boxes. Boxes of level 0 will be all organized into a vector and then stored on it. But when `vector_of_boxes` is first created by the type declaration, it is empty. The vector is filled by the following call to function `domainSplit`, which splits `domain_0` into `vector_of_boxes_0` given a preferred `box_size_0`.

This operation gives us a vector of boxes, but this is not yet a box layout. A box layout is obtained by associating boxes with processors. And so we define an initially empty `vector_of_processes_0`, which is then filled with MPI CPU numbers associated with appropriate boxes by the following call to `LoadBalance`. Finally, having in hand both vectors, i.e., the vector of boxes and the vector of processes, we can create the disjoint box layout and store it on `box_layout`. Before we leave this portion of the code we generate the data iterator for this box layout. We will use the data iterator to obtain portions of data associated with each box.

Now we are ready to define the data structures that correspond to fields  $E_z$ ,  $H_x$  and  $H_y$ . Here is how we do it for  $E_z$ :

```

#define NUMBER_OF_E_COMPONENTS 1

```

```

#define NG 1
...
int number_of_E_components = NUMBER_OF_E_COMPONENTS;
int number_of_ghost_cells = NG;
LevelData<FArrayBox> E_0(box_layout_0, number_of_E_components,
                        number_of_ghost_cells * IntVect::Unit);

```

`LevelData` is a class *template*, meaning that we have to provide additional type information in order to make it sufficiently specific. Here this additional type information is `FArrayBox`, which is a class that associates field data with a grid in a box. The field is assumed to be of type `Real`, which may evaluate to either `float` or `double` depending on how Chombo was configured, and it may have several components. In this case  $E_z$  has only one component. We also specify a number of ghost cells that should surround each box of the disjoint box layout. Observe that once you enlarge the boxes by adding the ghost cells to them the resulting box layout is no longer going to be disjoint. It is possible to extract both the smaller disjoint boxes and the larger overlapping boxes from a `LevelData` object.

Apart from defining `E_0`, which stands for “level 0 values of  $E_z$ ”, we are also going to define `E_0_0`, `E_0_1` and `E_0_2` to store  $E_z(t_0)$ ,  $E_z(t_0 + \Delta t_0)$  and  $E_z(t_0 + 2\Delta t_0)$  as all three will be needed for the interaction between level 0 and level 1 at every time step.

In a similar vain we also define:

```

#define NUMBER_OF_H_COMPONENTS 2
...
int number_of_H_components = NUMBER_OF_H_COMPONENTS;
LevelData<FArrayBox> H_0(box_layout_0, number_of_H_components,
                        number_of_ghost_cells * IntVect::Unit);

```

Here the number of components is 2, with the first component representing  $H_x$  and the second one  $H_y$ . We also define `H_0_0`, `H_0_1` and `H_0_2` to store  $\mathbf{H}(t_0 - \Delta t_0/2)$ ,  $\mathbf{H}(t_0 + \Delta t_0/2)$  and  $\mathbf{H}(t_0 + 3\Delta t_0/2)$  respectively. These will be needed for the interaction between level 0 and level 1 too.

## 5.2 Preiteration Procedures

Once the data structures for  $\mathbf{E}$  and  $\mathbf{H}$  fields have been created, they are initialized to zero.

Then we create additional Chombo vectors for storing labels and times. The vectors are initially empty. Every time a Gnuplot data file is written a numerical label pertaining to this file, as well as time that corresponds to this data snapshot are pushed onto either a `vector_of_labels` or a `vector_of_times`. There are separate vectors of labels and times for `E_0`, `E_1`, `H_0` and `H_1`, since these are all going to be dumped at various times and with various labels. A label is used in naming the data file. It is then used again in invoking the data file within the Gnuplot command file. Times that correspond to the data files are used in Gnuplot titles.

Apart from writing Gnuplot data files for  $\mathbf{E}$  and  $\mathbf{H}$  we are also going to write data files for tags and for level 1 boxes. These have their own vectors of labels too.

Finally, we introduce variables that will collect and store the minimum and maximum values of  $E_z$ ,  $H_x$  and  $H_y$ . These will then be used in scaling the plots. There is a separate time counter for level 0 called `time_0` and additional two separate time counters for  $E_z$  at level 1 and for  $\mathbf{H}$  at level 1

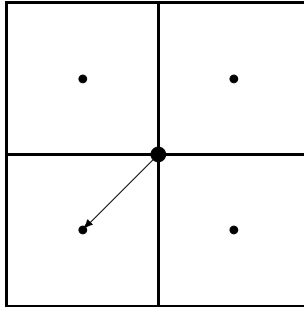


Figure 13: When a 2-dimensional  $(0,0)$  cell is refined with the refinement ratio of 2, it is split into 4 squares. The center of the lower-left square becomes now the beginning of the new refined grid. If the  $(x,y)$  coordinates of the first level 0 cell-centered grid point were  $(0,0)$  the  $(x,y)$  coordinates of the first level 1 cell-centered grid point are now going to be  $(-0.25, -0.25)$  assuming that the level 0 grid constant was 1.

called `time_E_1` and `time_H_1` correspondingly. All times are initialized to zero. Additionally we initialize:

```
Real ... dx_0 = 1.0, dt_0 = dx_0 / 2.0, x0_0 = 0.0, y0_0 = 0.0;
Real ... dx_1 = dx_0 / REFINERATIO, dt_1 = dx_1 / 2.0,
        x0_1 = x0_0 - dx_1 / 2.0, y0_1 = y0_0 - dx_1 / 2.0;
```

where `dx_0` is the grid constant at level 0 and `dt_0` is the level 0 time step. Similarly for level 1. We also have variables for the position of the left-lower corner of the grid. It is  $(0,0)$  at level 0 (`x0_0` and `y0_0`), but  $(-0.25, -0.25)$  at level 1. This shift must be taken into account when switching between the levels, otherwise non-physical results will be produced by the code. The reason for the shift is shown in figure 13.

### 5.3 The Iteration Loop

The iteration loop is the heart of the program. But its top level structure is very simple in the case of this demo. We know that it would be useless to make an excessive number of time steps, because we have no absorbing boundary conditions. As soon as the wave reaches the boundary, we must stop. This happens after `MAX_ITER` iterations and so the loop is:

```
#define MAX_ITER 50
...
for (int count = 0; count < MAX_ITER; count++)
{
    ...
}
```

Inside the loop we advance  $\mathbf{E}$  and  $\mathbf{H}$  at level 0 first, then check if we have a reason to generate level 1 and muck up with level 1 if we must.

The level 0 time advance pushes  $E$  first:

```
E_0.copyTo(E_0.interval(), E_0_0, E_0_0.interval());
time_E_1 = time_0;
for (data_iterator_0.reset(); data_iterator_0.ok();
    ++data_iterator_0)
    updateez_(E_0[data_iterator_0()].dataPtr(0),
              H_0[data_iterator_0()].dataPtr(0),
              H_0[data_iterator_0()].dataPtr(1),
              &(E_0[data_iterator_0()].loVect()[0]),
              &(E_0[data_iterator_0()].hiVect()[0]),
              &(E_0[data_iterator_0()].loVect()[1]),
              &(E_0[data_iterator_0()].hiVect()[1]),
              &time_0, &dt_0, &dx_0, &x0_0, &y0_0);
E_0.exchange(E_0.interval());
time_0 = time_0 + dt_0;
vector_of_times_E_0.push_back(time_0);
E_0.copyTo(E_0.interval(), E_0_1, E_0_1.interval());
E_0.copyTo(E_0.interval(), E_0_2, E_0_2.interval());
```

The first thing we do here is to save the current state of  $E_z(t_0)$  on  $E_{0\_0}$ . This is accomplished by invoking the `copyTo` method. The `interval` method returns the number of components. The `copyTo` method may be used to copy just one or two components, but in this case we want it to copy all there is – which happens to be just one (but we don’t have to know this). We are going to advance  $E_z$  from  $t_0$  ( $E_{0\_0}$ ) to  $t_0 + \Delta t_0$  ( $E_{0\_1}$ ) first, then we’re going to advance it to  $t_0 + 2\Delta t_0$  ( $E_{0\_2}$ ), because level 1 time step will need all three  $E_z$ s for its boundary conditions. Having saved  $E_z(t_0)$  on  $E_{0\_0}$  we save `time_0` on `time_E_1`, because this will be the time slice at which we will have to generate level 1  $E_z$ .

Now we advance  $E_z$ . This is done by looping over all the boxes of level 0 and calling a Fortran subroutine `updateez_` for each box. The subroutine’s Fortran declaration (taken from `update.F`) looks as follows:

```
subroutine updateez (Ez, Hx, Hy, imin, imax, jmin, jmax,
&    time, dt, dx, x0, y0)
implicit none
integer imin, imax, jmin, jmax
real*8 Ez(imin:imax, jmin:jmax)
real*8 Hx(imin:imax, jmin:jmax)
real*8 Hy(imin:imax, jmin:jmax)
real*8 time, dt, dx, x0, y0
```

Consummate Fortran programmers remember that Fortran passes all variables by reference, not by value. We must therefore obtain addresses of `Ez`, `Hx`, `Hy` and so on. These are obtained by:

1. Obtaining the C++ reference to the relevant field data (of type `FArrayBox`) in a given box, e.g.,

```
E_0[data_iterator_0()]
```

2. Invoking the `dataPtr` method, which belongs to class `FArrayBox` and which returns the address of the first data item that corresponds to a given component of the field. In our case the component number is 0, because  $E_z$  has one component only, so the full expression is

```
E_0[data_iterator_0()].dataPtr(0)
```

Chombo lays out its data in the Fortran fashion (column-major), so as to make passing to Fortran subroutines easier. This also serves us to remember that Fortran is *meant* to be used with Chombo – exactly as shown here.

Other variables that are passed to the subroutine are the integer coordinates of the lower-left corner of the box and of the upper-right corner of the box. These are obtained by calling `loVect` and `hiVect` methods on the `FArrayBox` object returned by `E_0[data_iterator_0()]`. Finally we have various scalars, `time_0`, `dt_0`, and so on, which are passed by using the standard C (and C++) address operator `&`.

The subroutine advances the field, and how it goes about it is a standard Fortran and FDTD stuff, so I won't elaborate on this here with the exception of drawing the reader's attention to one thing. The  $\mathbf{E}$  advance within the Fortran code is accomplished by the following nested DO loop:

```
do j = jmin + 1, jmax - 1
  do i = imin + 1, imax - 1
    Ez(i, j) = Ez(i, j)
    &          + 0.5 * (Hy(i, j) - Hy(i - 1, j))
    &          - Hx(i, j) + Hx(i, j - 1))
  end do
end do
```

Observe that we loop over the interior of the box only skipping its edges, i.e.,  $i = i_{\min}$ ,  $i = i_{\max}$ ,  $j = j_{\min}$  and  $j = j_{\max}$ . This is because the edges correspond to the ghost cells and we don't want to update these – there is no point, since such updates will be discarded anyway.

Where exactly do we look up the ghost cells? This happens when  $i = i_{\min}+1$  and when  $j = j_{\min}+1$ . For these values of  $i$  and  $j$  magnetic fields are collected from the ghost cells: `Hy(imin, j)` and `Hx(i, jmin)`. The ghost cells that correspond to  $i_{\max}$  and  $j_{\max}$ , on the other hand, are used in the  $\mathbf{H}$  update:

```
do j = jmin + 1, jmax - 1
  do i = imin + 1, imax - 1
    Hx(i, j) = Hx(i, j) + 0.5 * (Ez(i, j) - Ez(i, j + 1))
  end do
end do
do j = jmin + 1, jmax - 1
  do i = imin + 1, imax - 1
    Hy(i, j) = Hy(i, j) + 0.5 * (Ez(i + 1, j) - Ez(i, j))
  end do
end do
```

This is why we cannot carry out  $\mathbf{E}$  and  $\mathbf{H}$  updates at the same time. We must **exchange** the updated ghost cells data for  $\mathbf{E}$ , before we can update  $\mathbf{H}$ . This is accomplished by calling the `LevelData` method **exchange**. It can be used to exchange only a particular component, but here we exchange the whole field:

```
E_0.exchange(E_0.interval());
```

Time `time_0` is advanced explicitly by the C++ program outside of the Fortran field update subroutine.

Having updated  $E_z$  we save it on `E_0_1` (the previous state of  $E_z$  has already been saved on `E_0_0`) and we copy it to `E_0_2` too. We will then advance `E_0_2` to  $t_0 + 2\Delta t_0$ , but we will not advance `E_0` itself. `E_0` will instead be corrected by overwriting it with level 1 data if we find that we need to build level 1 to begin with.

The next step is to advance  $\mathbf{H}$ :

```
H_0.copyTo(H_0.interval(), H_0_1, H_0_1.interval());
time_0 = time_0 - dt_0 / 2.0;
time_H_1 = time_0;
for (data_iterator_0.reset(); data_iterator_0.ok();
    ++data_iterator_0)
    updateh_(E_0[data_iterator_0()].dataPtr(0),
             H_0[data_iterator_0()].dataPtr(0),
             H_0[data_iterator_0()].dataPtr(1),
             &(E_0[data_iterator_0()].loVect()[0]),
             &(E_0[data_iterator_0()].hiVect()[0]),
             &(E_0[data_iterator_0()].loVect()[1]),
             &(E_0[data_iterator_0()].hiVect()[1]),
             &time_0, &dt_0, &dx_0, &x0_0, &y0_0);
H_0.exchange(H_0.interval());
time_0 = time_0 + dt_0;
vector_of_times_H_0.push_back(time_0);
H_0.copyTo(H_0.interval(), H_0_2, H_0_2.interval());
```

The manipulations here should now be easy to understand. We save the current  $\mathbf{H}$  on `H_0_1`. At the end of the iteration loop `H_0_1` will be copied to `H_0_0` so that the latter will in effect end up storing the *previous* value of  $\mathbf{H}$ . Then we push our timer back by half a time step and remember this time on `time_H_1`, in case we have to build and advance level 1. Then we iterate over the boxes of level 0 again and call Fortran subroutine `updateh_` on every box. This subroutine will advance  $\mathbf{H}$ . After the subroutine returns we have to exchange the ghost cell data by calling

```
H_0.exchange(H_0.interval());
```

we advance the timer and save the new  $\mathbf{H}(t_0 + 3\Delta t_0/2)$  on `H_0_2`.

Having done this we advance  $E_z$  yet again so as to reach  $E_z(t_0 + 2\Delta t_0)$ :



```

time_0 = time_0 - dt_0 / 2.0;
for (data_iterator_0.reset(); data_iterator_0.ok();
    ++data_iterator_0)
    updateez_(E_0_2[data_iterator_0()].dataPtr(0),
              H_0[data_iterator_0()].dataPtr(0),
              H_0[data_iterator_0()].dataPtr(1),
              &(E_0_2[data_iterator_0()].loVect()[0]),
              &(E_0_2[data_iterator_0()].hiVect()[0]),
              &(E_0_2[data_iterator_0()].loVect()[1]),
              &(E_0_2[data_iterator_0()].hiVect()[1]),
              &time_0, &dt_0, &dx_0, &x0_0, &y0_0);
E_0_2.exchange(E_0_2.interval());

```

Observe that we don't touch  $E_0$ . Instead it is  $E_{0,2}$  that gets pushed. We don't advance the clock after the subroutine returns either, because we want the clock to stay at  $t_0 + \Delta t_0$  in preparation for the next iteration. After this time step  $E_{0,2}$  will be discarded.

This completes the level 0 advance.

## 5.4 Tagging Cells of Level 0

Now we have data for  $E$  and  $H$  for six different time slices – including one that is quite ahead, i.e.,  $t_0 + 2\Delta t_0$ . We use this data to tag level 0 cells for refinement. This is done by

```

IntVectSet tag_set_0;
tag_cells(tag_set_0, E_0_0, E_0_1, E_0_2, H_0_0, H_0_1, H_0_2);

```

The first line declares and creates the set of tags, `tag_set_0`. The second line calls my own function, which goes through every cell of level 0 and looks at the 6 values of the fields that are passed to it. We can now tag cells where the values of the fields exceed a certain threshold and this is what the following version of function `tag_cells` does:

```

void tag_cells(IntVectSet &tag_set,
              LevelData<FArrayBox> &E_0,
              LevelData<FArrayBox> &E_1,
              LevelData<FArrayBox> &E_2,
              LevelData<FArrayBox> &H_0,
              LevelData<FArrayBox> &H_1,
              LevelData<FArrayBox> &H_2)
{
    DisjointBoxLayout box_layout = E_0.disjointBoxLayout();
    DataIterator data_iterator(box_layout);

    Real diff_threshold2, value_threshold2;
    int tag_on_diffs = FALSE;
    int tag_on_values = FALSE;
    ParmParse parameter_parser;

```

```

parameter_parser.query("tag_on_diffs", tag_on_diffs);
parameter_parser.query("tag_on_values", tag_on_values);

diff_threashold2 = DIFFS_TAG_THRESHOLD * DIFFS_TAG_THRESHOLD;
value_threashold2 = VALUE_TAG_THRESHOLD * VALUE_TAG_THRESHOLD;

tag_set.makeEmpty();

for (data_iterator.reset(); data_iterator.ok(); ++data_iterator)
{
    BoxIterator box_iterator(E_0[data_iterator()].box());
    for (box_iterator.reset(); box_iterator.ok(); ++box_iterator)
    {
        IntVect cell = box_iterator();
        Box cell_box(cell, cell);
        Real Ez_0 = E_0[data_iterator()].get(cell, 0);
        Real Hx_0 = H_0[data_iterator()].get(cell, 0);
        Real Hy_0 = H_0[data_iterator()].get(cell, 1);
        Real Ez_1 = E_1[data_iterator()].get(cell, 0);
        Real Hx_1 = H_1[data_iterator()].get(cell, 0);
        Real Hy_1 = H_1[data_iterator()].get(cell, 1);
        Real Ez_2 = E_2[data_iterator()].get(cell, 0);
        Real Hx_2 = H_2[data_iterator()].get(cell, 0);
        Real Hy_2 = H_2[data_iterator()].get(cell, 1);

        if(tag_on_diffs)
        {
            Real dEz_0 = Ez_1 - Ez_0;
            Real dEz_1 = Ez_2 - Ez_1;
            Real dHx_0 = Hx_1 - Hx_0;
            Real dHx_1 = Hx_2 - Hx_1;
            Real dHy_0 = Hy_1 - Hy_0;
            Real dHy_1 = Hy_2 - Hy_1;

            if (((dEz_0 * dEz_0) > diff_threashold2)
                || ((dHx_0 * dHx_0) > diff_threashold2)
                || ((dHy_0 * dHy_0) > diff_threashold2)
                || ((dEz_1 * dEz_1) > diff_threashold2)
                || ((dHx_1 * dHx_1) > diff_threashold2)
                || ((dHy_1 * dHy_1) > diff_threashold2))
                tag_set |= cell_box;
        }

        if(tag_on_values)
        {
            if (((Ez_0 * Ez_0) > value_threashold2)
                || ((Hx_0 * Hx_0 + Hy_0 * Hy_0) > value_threashold2))

```

```

        || ((Ez_1 * Ez_1) > value_threashold2)
        || ((Hx_1 * Hx_1 + Hy_1 * Hy_1) > value_threashold2)
        || ((Ez_2 * Ez_2) > value_threashold2)
        || ((Hx_2 * Hx_2 + Hy_2 * Hy_2) > value_threashold2))
    tag_set |= cell_box;
}
}
}
return;
}

```

We assume that all fields live on the same disjoint box layout. It is therefore sufficient to extract the box layout from the first field and then use it for others. This is done by invoking the `disjointBoxLayout` method of the `LevelData` class. We can then associate the data iterator with the box layout, the same way we did it when advancing fields of level 0.

We clear the set of tags by invoking the `makeEmpty` method.

Now we iterate over the boxes of `LevelData`. For each box returned by `E_0[data_iterator()].box()` we create the box iterator. The box iterator simply returns integer coordinates of every point in the box. With these in hand, as we iterate over the box, we can retrieve the values of  $E_z$ ,  $H_x$  and  $H_y$  for every point within the box by calling the `FArrayBox::get` method, which takes the point and the number of the component as its arguments.

Now we can check if any of the numbers returned exceeds the predetermined threshold (the `tag_on_values` option). If the threshold is exceeded by any of the fields, we add the corresponding cell to the set of tags.

The other option (`tag_on_diffs`) looks at how much the fields change in time, since this is directly related to how much they change in space as well on account of Maxwell equations. It is where they change most that we want to have a higher resolution, both in space and in time.

If the set of tags is *not* empty, it means that we have to refine our grid, build level 1 data, and then advance it. This is going to be a lot of work, of course, both to code (which means a lot of work for me) and to compute (which means a lot of work for computer).

The whole level 1 part of the code is enclosed within the following `if` statement:

```

if (! tag_set_0.isEmpty()) {
...
... <a lot of hair raising stuff here> ...
...
} // of if (! tag_set_0.isEmpty())

```

## 5.5 Constructing the Level 1 Grid

In order to construct level 1 we have to

1. generate a vector of level 1 boxes that would cover the tagged cells of level 0;
2. convert the vector into a disjoint box layout, i.e., associate MPI process numbers with the boxes;
3. sow  $\mathbf{E}$  and  $\mathbf{H}$  fields on the level 0 disjoint box layout by interpolating data from level 1.

The level 1 boxes are generated by a special device that applies a Berger-Rigoutsos algorithm to the tagged cells and boxes of level 0. The newly generated boxes of level 1 are nested within the level 0 boxes in a special way that ensures correct data transfers between the two levels. The device is implemented as a Chombo class `BRMeshRefine`. It is used in two stages

1. we have to create and initialize the device first;
2. then we use the device by invoking a method called `regrid`.

The `regrid` method works on multiple Chombo levels all at the same time. The levels are represented by a *vector of vectors of boxes*, each entry in the vector (which is a vector of boxes) representing a level. We have to specify which levels we want to refine and what refinement ratios we want to use. What comes out is a *new vector of vectors of boxes*, which can be then used to generate disjoint box layouts for each level. Once the layouts have been generated, various fields can be sown on them by interpolating their values from coarser levels.

In the case of this demo it is all going to be much simpler, because we only have two levels, but the machinery is the same.

Here is what this part of the code looks like:

```
#define REFINE_RATIO                2
#define REFINE_FILL_RATIO           1.0
#define REFINE_BLOCK_FACTOR         4
#define REFINE_BUFFER_SIZE          1
#define REFINE_MAX_SIZE              0
#define REFINE_BASE_LEVEL           0
#define REFINE_TOP_LEVEL             0
#define INTERPOLATION_RADIUS        1
...

Vector< Vector<Box> > old_vector_of_vectors_of_boxes,
    new_vector_of_vectors_of_boxes;

old_vector_of_vectors_of_boxes.clear();
new_vector_of_vectors_of_boxes.clear();
old_vector_of_vectors_of_boxes.push_back(vector_of_boxes_0);

Vector<IntVectSet> vector_of_tag_sets;
vector_of_tag_sets.clear();
vector_of_tag_sets.push_back(tag_set_0);

int refine_ratio = REFINE_RATIO;
Vector<int> vector_of_refinements;
vector_of_refinements.clear();
vector_of_refinements.push_back(refine_ratio);

Real refine_fill_ratio = REFINE_FILL_RATIO;
int refine_block_factor = REFINE_BLOCK_FACTOR;
int refine_buffer_size = REFINE_BUFFER_SIZE;
int refine_max_size = REFINE_MAX_SIZE;
```

```

int refine_base_level = REFINES_BASE_LEVEL;
int refine_top_level = REFINES_TOP_LEVEL;

BRMeshRefine mesh_refine(domain_0, vector_of_refinements,
                        refine_fill_ratio, refine_block_factor,
                        refine_buffer_size, refine_max_size);
mesh_refine.regrid(new_vector_of_vectors_of_boxes,
                  vector_of_tag_sets,
                  refine_base_level,
                  refine_top_level,
                  old_vector_of_vectors_of_boxes);

```

We begin by creating the two crucial vectors:

- `old_vector_of_vectors_of_boxes`,
- `new_vector_of_vectors_of_boxes`.

Both vectors are cleared and then we *push* `vector_of_boxes_0`, i.e., the vector of boxes that represents level 0, onto the `old_vector_of_vectors_of_boxes`. Chombo vectors are similar to stacks and the `push_back` method simply appends an item to the vector extending its length at the same time.

Then we create a `vector_of_tag_sets`. We have only one tag set here, the one generated for level 0, and we push it onto the vector.

Finally we have to create a `vector_of_refinements`. Each entry in this vector specifies the refinement between this level and the next level.

Then we have to initialize a number of scalar parameters that are used to steer the refinement procedure (e.g., they can be used to determine the smallest size of a newly generated box). Here I mostly adhered to the defaults suggested in the Design Document. Playing with the value of `REFINE_BLOCK_FACTOR` showed that the `regrid` method can be easily broken if the numbers are not set correctly (a run-time error is generated). For example, if the smallest upper level box is  $16 \times 16$  then we cannot request `REFINE_BLOCK_FACTOR` of, say, 5. On the other hand allowing for boxes of too small a size to be generated within the refined level may lead to severe computational inefficiencies. Generally, the larger the boxes, the better.

Eventually with everything in place we create the device, `mesh_refine`, and then use it to generate the new level 1 vector of boxes that is going to be the second entry in the `new_vector_of_vectors_of_boxes`. The first entry will remain `vector_of_boxes_0`:

```

Vector<Box>& vector_of_boxes_1 = new_vector_of_vectors_of_boxes[1];

```

Now we go through the similar procedure to the one we deployed to generate the disjoint box layout for level 0:

```

Vector<int> vector_of_processes_1;
LoadBalance(vector_of_processes_1, vector_of_boxes_1);
DisjointBoxLayout
    box_layout_1(vector_of_boxes_1, vector_of_processes_1);
...

```

```
DataIterator data_iterator_1(box_layout_1);
```

Figures 3 and 4 illustrate this procedure. We begin with a set of tagged cells at level 0 at  $t = 20.5$  and end up with a level 1 grid that is organized into boxes and distributed amongst the MPI processes. Figure 6 shows both the tagged cells and the level 1 grid points in a close-up.

## 5.6 Filling the Level 1 Grid With Data

Now we have to create fields for level 1 and fill them with data. The field is an object of type `LevelData` that is created the same way as for level 0. But here we must also associate an interpolator with it, so that we can fill it with data interpolated from level 0:

```
LevelData<FArrayBox> E_1(box_layout_1, number_of_E_components,
                          number_of_ghost_cells * IntVect::Unit);
FineInterp E_interpolator(box_layout_1, number_of_E_components,
                          vector_of_refinements[0], domain_1);
```

The device that does the interpolation is an object of class `FineInterp`. When creating this object we must specify the box layout it is associated with, the number of field components we are going to interpolate, the refinement between this and the coarser level and the domain the current box layout fits in. The device so created is then invoked with the `interpToFine` method that is going to interpolate level 0 data and fill all level 1 boxes with it.

This device will *not* fill the level 1 ghost cells though. For this we need to use another device, which is called `PiecewiseLinearFillPatch`, and which has to be specified for the ***E*** field too:

```
PiecewiseLinearFillPatch
E_fill_patch(box_layout_1, box_layout_0, number_of_E_components,
             domain_1, vector_of_refinements[0],
             interpolation_radius);
```

This device fills only the ghost cells of level 1 with data interpolated from level 0, but it will interpolate the data *both* in space and in time, in order to provide data for level 1 boundary conditions at every level 1 time step. The `FineInterp` class does *not* interpolate in time.

Now we repeat the same procedure to generate `LevelData` and the interpolators for ***H***:

```
LevelData<FArrayBox> H_1(box_layout_1, number_of_H_components,
                          number_of_ghost_cells * IntVect::Unit);
FineInterp H_interpolator(box_layout_1, number_of_H_components,
                          vector_of_refinements[0], domain_1);
PiecewiseLinearFillPatch
H_fill_patch(box_layout_1, box_layout_0, number_of_H_components,
             domain_1, vector_of_refinements[0],
             interpolation_radius);
```

With these two interpolation devices in hand we can now fill the `LevelData` fields with data. And this is how we do it for ***E***:

```
E_interpolator.interpToFine(E_1, E_0_0);
if (reuse_level_1_data && have_valid_level_1_E_data)
    E_1_save.copyTo(E_1_save.interval(), E_1, E_1.interval());
E_1.exchange(E_1.interval());
E_fill_patch.fillInterp(E_1, E_0_0, E_0_0, 0.0,
                        start_source_component,
                        start_destination_component,
                        number_of_E_components);
...
```

The first line fills the entire ***E***<sub>1</sub> with level 0 data interpolated from ***E***<sub>0\_0</sub>. The following `if` statement overwrites a portion of ***E***<sub>1</sub> with saved level 1 data, if such is available, from the previous iteration. Only the portion of level 1 data from the previous grid that overlaps with the current grid will be copied. Then we attend to the ghost cells by exchanging the data between level 1 boxes, and by filling ghost boxes on the coarse/fine boundary with data interpolated from level 0.

The procedure is more complicated for the ***H*** field though. This time we have to time-interpolate data between ***H***( $t_0 - \Delta t_0/2$ ) and ***H***( $t_0 + \Delta t_0/2$ ) first, as shown in figure 11. I have written my own function for doing the time-interpolation called `time_interpolate`, which has a very similar synopsis to the `fillInterp` method of the `PiecewiseLinearFillPatch` class, but it interpolates *within* the level instead. Then the time-interpolated data is space interpolated from level 0 to level 1 using `interpToFine`:

```
LevelData<FArrayBox> H_0_x(box_layout_0, number_of_H_components,
                           number_of_ghost_cells * IntVect::Unit);
time_interpolate(H_0_x, H_0_0, H_0_1, 0.75,
                 start_source_component,
                 start_destination_component,
                 number_of_H_components);
H_interpolator.interpToFine(H_1, H_0_x);
H_1.exchange(H_1.interval());
H_fill_patch.fillInterp(H_1, H_0_0, H_0_1, 0.75,
                        start_source_component,
                        start_destination_component,
                        number_of_H_components);
```

Observe that in both cases we call the `exchange` method of the `LevelData` class prior to calling `fillInterp`. The reason for this is that `fillInterp` fills the ghost cells on the coarse/fine boundary only. The ghost cells within the level 1 grid, which are not on the coarse/fine level boundary have to be filled by calling `exchange`, the same way we did it for the level 0 computations.

This demonstration code does not reuse level 1 data for ***H***<sub>1</sub>, because it is never readily available and it would have to be time-interpolated. However, level 1 data reuse both for ***E***<sub>1</sub> and for ***H***<sub>1</sub> should be easy to implement for  $\Delta t_1 = \Delta t_0/3$ .

## 5.7 Advancing the Level 1 $E$

Now we have the level 1 grid constructed, the field data structures declared and filled and we are ready to advance the level 1  $E$ . This is done very similarly to the level 0 advance. The only difference is that after every time step we have to call the `fillInterp` method in order to fill the coarse/fine boundary with new data that is time and space interpolated from level 0. But we invoke the `exchange` method too, because we have to exchange the ghost data between the level 1 boxes as well:

```
for (data_iterator_1.reset(); data_iterator_1.ok();
    ++data_iterator_1)
    updateez_(E_1[data_iterator_1()].dataPtr(0),
              H_1[data_iterator_1()].dataPtr(0),
              H_1[data_iterator_1()].dataPtr(1),
              &(E_1[data_iterator_1()].loVect()[0]),
              &(E_1[data_iterator_1()].hiVect()[0]),
              &(E_1[data_iterator_1()].loVect()[1]),
              &(E_1[data_iterator_1()].hiVect()[1]),
              &time_E_1, &dt_1, &dx_1, &x0_1, &y0_1);
time_E_1 = time_E_1 + dt_1;
E_1.exchange(E_1.interval());

// At this time E_1 is exactly in the middle between
// E_0_0 and E_0_1. We interpolate ghost cells from
// both then.

E_fill_patch.fillInterp(E_1, E_0_0, E_0_1, 0.50,
                        start_source_component,
                        start_destination_component,
                        number_of_E_components);
```

Then we advance the magnetic field  $H$  similarly:

```
time_E_1 = time_E_1 - dt_1 / 2.0;
for (data_iterator_1.reset(); data_iterator_1.ok();
    ++data_iterator_1)
    updateh_(E_1[data_iterator_1()].dataPtr(0),
              H_1[data_iterator_1()].dataPtr(0),
              H_1[data_iterator_1()].dataPtr(1),
              &(E_1[data_iterator_1()].loVect()[0]),
              &(E_1[data_iterator_1()].hiVect()[0]),
              &(E_1[data_iterator_1()].loVect()[1]),
              &(E_1[data_iterator_1()].hiVect()[1]),
              &time_E_1, &dt_1, &dx_1, &x0_1, &y0_1);
time_E_1 = time_E_1 + dt_1;
H_1.exchange(H_1.interval());

// At this time H_1 is 0.75 * H_0_1 + 0.25 * H_0_2. We
```



```

// interpolate on the ghost cells accordingly. 0.25 means
// that H_1 is *closer* to H_0_1.

H_fill_patch.fillInterp(H_1, H_0_1, H_0_2, 0.25,
                        start_source_component,
                        start_destination_component,
                        number_of_H_components);

```

And finally we advance  $\mathbf{E}$  yet again so as to get it to  $t_0 + \Delta t_0$  and match  $\mathbf{E}(t_0 + \Delta t_0)$  of level 0:

```

time_E_1 = time_E_1 - dt_1 / 2.0;
for (data_iterator_1.reset(); data_iterator_1.ok();
    ++data_iterator_1)
    updateez_(E_1[data_iterator_1()].dataPtr(0),
              H_1[data_iterator_1()].dataPtr(0),
              H_1[data_iterator_1()].dataPtr(1),
              &(E_1[data_iterator_1()].loVect()[0]),
              &(E_1[data_iterator_1()].hiVect()[0]),
              &(E_1[data_iterator_1()].loVect()[1]),
              &(E_1[data_iterator_1()].hiVect()[1]),
              &time_E_1, &dt_1, &dx_1, &x0_1, &y0_1);
time_E_1 = time_E_1 + dt_1;
E_1.exchange(E_1.interval());

// At this time E_1 is exactly on the same time slice
// as E_0_1. We interpolate on the ghost cells from
// E_0_1 alone then.

E_fill_patch.fillInterp(E_1, E_0_1, E_0_1, 0.0,
                        start_source_component,
                        start_destination_component,
                        number_of_E_components);

```

At the end of this double time step we save  $\mathbf{E}_1$  for a possible future use, i.e., if a new fine level is going to be generated on the next coarse level time step:

```

E_1_save.define(box_layout_1, number_of_E_components,
                number_of_ghost_cells * IntVect::Unit);
E_1.copyTo(E_1.interval(), E_1_save, E_1_save.interval());
have_valid_level_1_E_data = TRUE;

```

## 5.8 Correcting the Level 0 $\mathbf{E}$

The level 1  $\mathbf{E}(t_0 + \Delta t_0)$  obtained in this way should be more accurate than the level 0  $\mathbf{E}(t_0 + \Delta t_0)$ , because we have advanced it on a finer grid and used a shorter time step. We are therefore going to

replace the level 0  $\mathbf{E}(t_0 + \Delta t_0)$  with data obtained by averaging the level 1  $\mathbf{E}(t_0 + \Delta t_0)$  data in the parts of the level 0 grid that overlap with the level 1 grid. This is done by a special device of class `CoarseAverage`, which is used as follows:

```
CoarseAverage E_averager(box_layout_1, box_layout_0,
                        number_of_E_components,
                        vector_of_refinements[0]);
E_averager.averageToCoarse(E_0, E_1);
```

The declaration must specify the disjoint box layouts at both levels as well as the number of the field components and the refinement ratio. Then the averaging and the data transfer is accomplished by calling the `averageToCoarse` method of the class.

This completes the fine level  $\mathbf{E}$  advance. There is one more thing only that has to be done and that is highly non-trivial and I am going to talk about it briefly in section 6.

The  $\mathbf{H}$  field is advanced similarly. We have to prepare the initial condition for it first, then we advance the field and push it back onto level 0.

## 5.9 Generating Gnuplot Data Files

Gnuplot data files are written throughout the advance so that field evolution can be observed at both levels. I have constructed my own function `write_level_data` that takes an object of class `LevelData` and writes its components on plain text files in a format readable by the Gnuplot `splot` function.

This is based on the obvious assumption that the data can be collected. This assumption is going to break though, when the program is run under MPI on a CPU farm. In fact this program, as it is presented in this report, cannot be run on a farm, even though the basic data structures and the basic composition of the program prepare it for distribution over multiple CPUs. The reason for this is that iterations over `LevelData` boxes return only the boxes that are associated with a given MPI process. Chombo does not provide means for exchange of data between CPUs other than the `exchange` method of the `LevelData` class. Consequently my data collecting function will not work on a parallel processor.

When we get to running Chombo jobs on multiple CPUs we will have to switch to using Chombo IO, which is based on HDF5. HDF5 data files cannot be viewed with Gnuplot. We will have to develop our own facilities for extracting relevant data from the Chombo HDF5 files and viewing them in a way that is similar to what I have demonstrated in this report.

## 5.10 Reading the Input File

The execution of the program can be customized by setting a number of logical parameters on an input file, which is read at the very beginning by the `parameter_parser` object of class `ParmParse`. The following parameters are defined:

**tag\_on\_diffs** Chombo cells are tagged if field derivatives exceed a certain threshold. 1 means “on”, 0 means “off”.

**tag\_on\_values** Chombo cells are tagged if field values exceed a certain threshold. “on” and “off” as above.

**advance\_e1** Level 1 electric field is constructed and advanced if there are any tagged cells on level 0.

**advance\_h1** Level 1 magnetic field is constructed and advanced if there are any tagged cells on level 0.

**correct\_e0** Level 0 electric field is corrected by averaging level 1 data if available.

**correct\_h0** Level 0 magnetic field is corrected by averaging level 1 data if available.

**reuse\_level\_1\_data** Previous step level 1 data is reused when available and valid for generation of the level 1 electric and/or magnetic field at the beginning of the next level 1 time step.

**hidden3d** The gnuplot command file is prepared for the display of data using hidden line removal. This is very time consuming and the resulting animations are very slow. For this reason only a quarter of the domain is displayed.

**verbose** The program runs in a verbose mode. It prints very little at present and clumsily to boot. Used for minor debugging only.

Here is an example of an input file `fdtd.input`:

```
tag_on_diffs = 1
tag_on_values = 1
advance_e1 = 1
advance_h1 = 1
correct_e0 = 1
correct_h0 = 1
reuse_level_1_data = 1
hidden3d = 0
verbose = 0
```

## 6 Refluxing

As the fine level data is advanced there is something that flows away from the fine region and into the coarse region. This something has been crudely approximated by the coarse level computations. There is also something that flows from the coarse level and into the fine level. This something is represented by filling the fine level ghost cells with data interpolated in time and in space from the coarse level. But the stuff that flows from the fine level into the coarse level is basically discarded in the current demonstration program.

This can be seen also by focusing on what happens to the ghost cells data of level 1. This data is exchanged between the level 1 boxes, but if a given side of a level 1 box doesn't have any level 1 neighbours, the data is discarded instead of, say, being passed to the level 0 ghost cells adjacent to this box – this, because there are no level 0 ghost cells there, and because this level 0 time step has already been executed.

Chombo provides special machinery in the form of the `LevelFluxRegister` class for capturing the stuff, accumulating it through the fine level time stepping procedure, and then spilling it onto the coarse level. The spilling itself must be done with caution too, because some of this stuff is already there, since it's been computed on the coarse level. What we really want to do is to compare the coarse level “flux” with the fine level “flux” on the coarse/fine boundary and *correct* the coarse level flux if need be. This is what I have called “spilling”. We don't really spill all that we have accumulated through the fine level time step. We only spill the excess, so to speak, and even this spilling may be moderated by weighting between the coarse level and the fine level “fluxes”.



This procedure is called *refluxing*. The current demo doesn't do it yet. As can be seen from the animations, this lack of refluxing doesn't seem to affect the evolution of the fields visibly, but this is because our computations are not very accurate and because the missing flux corrections are probably very small.

But refluxing is essential for AMR and we will have to work out how to do it in the FDTD context. It is not obvious what should be refluxed. Should we look at  $\mathbf{E} \times \mathbf{H}$ ? If so, how should we use it to make specific corrections to  $\mathbf{E}$  and  $\mathbf{H}$  at the coarse/fine boundary? It is even less obvious how refluxing should be carried out in combination with the leapfrog time stepping.

This single issue ought to be resolved and thoroughly tested, but it doesn't really have to halt all further development. It is quite OK to construct and test example codes without this feature and then add it, once we know how to do this.

## 7 A Multilevel System

The program presented in this report is a very simple 2-level AMR application. It can be easily extended to 3 or more levels, although at a highly increasing cost in terms of program complexity. The Chombo class `AMR` is specially designed to facilitate construction of the multi-level logic in AMR programs. This class, however, can be used only when everything else is in place, fully understood and fully tested. We will have to figure out and test the refluxing problem, and we will have to have in place utilities for viewing and analyzing Chombo HDF5 output first.

My preference is for managing AMR levels manually, as I have shown in this program, up to perhaps 3 levels deep, and using such a framework to construct and run tests of the method in the FDTD context. Once we have everything ironed out, fully understood, debugged, and ready to go, then will be the right time to pack it all into the `AMR` class and deliver an FDTD/Chombo production environment.



# References

- [1] Chombo: see <http://seesar.lbl.gov/anag/chombo/>
- [2] P. Colella, D. T. Graves, T. J. Ligoeki, D. F. Martin, D. Modiano, D. B. Serafini, B. Van Straalen, “Chombo Software Package for AMR Applications Design Document”, Applied Numerical Algorithms Group, NERSC Division, Lawrence Berkeley National Laboratory, Berkeley, CA, September 12, 2003
- [3] Gnuplot: see <http://www.gnuplot.info/>
- [4] HDF5: see <http://hdf.ncsa.uiuc.edu/HDF5/>
- [5] Dennis M. Sullivan, “Electromagnetic Simulation Using the FDTD Method”, IEEE Press Series on RF and Microwave Technology, IEEE Press, New York, 2000, ISBN 0-7803-4747-1
- [6] Allen Taflove, Susan C. Hagness, “Computational Electrodynamics”, Second Edition, Artech House, Boston, London, 2000, ISBN 1-58053-076-1

# Index

- 3-dimensional simulations, 3
- absorbing boundary conditions, 7
- AMR, 3
  - time step, 14
- ANAG, 3
- animations, 14
- Berger-Rigoutsos algorithm, 28
- C++ code
  - basic structure, 17
  - iteration, 17
- cell tagging, 10
- Chombo, 3
  - AMR, 36
  - BoxIterator, 27
  - BRMeshRefine, 28
    - regrid, 28
  - C++ and Fortran, 3
  - CoarseAverage, 34
    - averageToCoarse, 34
  - code dependencies, 4
  - compilation and linking, 6
  - Cygwin, 4
  - Design Document, 4
  - directory tree, 4
  - domainSplit, 19
  - FArrayBox, 20
    - box, 27
    - dataPtr, 23
    - get, 27
    - hiVect, 23
    - loVect, 23
  - FineInterp, 30
    - interpToFine, 30
  - GNUmakefile, 5
  - IntVect, 19
    - Unit, 19
    - Zero, 19
  - IntVectSet, 25
    - makeEmpty, 27
  - IO, 3
  - LevelData, 8, 20, 30
    - copyTo, 22
    - disjointBoxLayout, 27
    - exchange, 24, 31, 34
    - interval, 22
  - LevelFluxRegister, 35
    - reflux, 36
  - libraries, 5
  - LoadBalance, 19
  - MacOS, 4
  - naming convention, 5
  - NodeFArrayBox, 8
  - ParmParse, 34
    - query, 25
  - PiecewiseLinearFillPatch, 30
  - ProblemDomain, 19
  - Real, 20
  - Reference, 4
  - refine, 19
  - refluxing, 35
  - staggered grid, 8
    - and DisjointBoxLayout, 8
  - tests and examples, 5
  - Vector, 19
    - push\_back, 29
    - visualization, 3
- cylindrical wave, 10
- FDTD, 3
- Gaussian pulse, 9
- Gnuplot, 3
  - command files, 3
  - data files, 3
  - splot, 34
- HDF5, 4
- LBL, 3
- level 0
  - $E_z$ , 20
  - $\mathbf{H}$ , 20
  - and level 1, 20
  - disjoint box layout, 19
  - domain, 18
  - vector of boxes, 19
  - vector of processes, 19
- Maxwell equations, 7
  - divergence equations, 7

- overlying data, 14
- parallel computing, 3
- Perl, 4
- refinement ratio, 10
- refluxing, 14, 36
- staggered grid, 7
- tag\_cells, 25
- time step
  - leapfrog, 14
  - time padding, 16
  - two level, 16
- time\_interpolate, 31
- write\_level\_data, 34